A Model Driven Reverse Engineering Framework for Extracting Business Rules out of a Java Application

Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, Jacques Perronnet

A Model Driven Reverse Engineering Framework for Extracting Business Rules out of a Java Application

Valerio Cosentino[1,2], Jordi Cabot[1], Patrick Albert[2], Philippe Bauquel[3] and Jacques Perronnet[3]

[1] AtlanMod, INRIA & EMN, Nantes, France
[2] IBM CAS France
[3] IBM, France
{albertpa, bauquel.p, jacques_perronnet, valerio.cosentino}@fr.ibm.com, jordi.cabot@mines-nantes.fr

Abstract. In order to react to the ever-changing market, every organization needs to periodically reevaluate and evolve its company policies. These policies must be enforced by its Information System (IS) by means of a set of business rules that drive the system behavior and data. Clearly, policies and rules must be aligned at all times but unfortunately this is a challenging task. In most ISs implementation of business rules is scattered among the code so appropriate techniques must be provided for the discovery and evolution of evolving business rules.
In this paper we describe a model driven reverse engineering framework aiming at extracting business rules out of Java source code. The use of modeling techniques facilitate the representation of the rules at a higher-abstraction level which enables stakeholders to understand and manipulate them.

1 Introduction

Today market needs oblige organizations to change periodically their policies expressed as a set of business rules. A business rule represents a relevant action or procedure aiming at defining or constraining some precise aspect of a business. Business rules are a key component of the Information System (IS) of the company. Unfortunately, they are not usually implemented as a single and easily identifiable component in the IS but are generally scattered in many parts of the IS source code. This makes it very difficult to quickly and safely evolve the organizational policies.

To tackle this issue we propose a new Business Rule Extraction (BREX) framework. BREX [1] is the process of extracting business rules out of an IS, isolating the code segments which are directly related to business processes. BREX includes three major activities: Variable Classification, Business Rule Identification (mainly based on Program Slicing [2] techniques) and Business Rule Representation.

Variable Classification is used to reduce the number of variables to analyse. It aims at finding variables that represent domain/business concepts and hint at business rules. The set of domain concepts represent the sphere of non-technical knowledge embedded in the application.

Business Rule Identification aims at identifying business rules by slicing the source code [3] to focus on the chunks of code that are relevant to the domain variables, identified in the previous step. A set of chunks related to the same variable conforms a business rule.

Business Rule Representation consists of presenting the extracted business rules through artifacts (graphs, text, ...) amenable to human comprehension.

A further activity in BREX is the traceability of business rules from the source code. This task is not always present in BREX frameworks but we believe is a key component to explain and justify the origin of the extracted business rules.

In this sense, this paper describes a model-driven framework for extracting business rules out of a Java application. We show that Model Driven Engineering (MDE) applied to reverse engineering/BREX approaches offers some important benefits with respect to previous works. MDE allows working on an abstract homogeneous representation of the system that avoids technological problems and provides a non-intrusive solution, since the extraction process is performed by working with the model of the system and not the system itself. Moreover, when representing the system as a model we can benefit from the plethora of MDE tools available to manipulate the (model of the) system.

MDE allows us to build a framework composed by independent and modular steps, since each of them is related to the others by its input and output models. In this way, the user can stop the BREX process at any moment, selecting the level of output that better fits his needs.

The framework is fully automatic but allows user intervention at the end of each sub-step. This allows users to complement the automatic process in order to refine and improve the results of our extraction heuristics, e.g. users could provide information about the company "coding style" to facilitate the identification of rules.

This paper is structured as follows: Section 2 presents a running example; Section 3 introduces the overall approach; Section 4 illustrates Traceability in the framework, which describes how the entities composing the artifacts in the framework are related to the source code; Section 5 analyses the result of this framework; Section 6 discusses the related work and Section 7 closes the paper with conclusion and future work.

2 Running Example

In order to illustrate our framework, we will use as running example a Java application that belongs to the simulation software category and that contains several business rules.

The application simulates the behavior of animals and humans in a meadow, where each actor, animal or human, can act and move according to its nature. Two different functionalities are implemented in this application: one represents the business logic and describes how predator-prey interactions affect population sizes. The second one is used to store statistical information about the actors participating in the simulation.

A schema of the application classes and their relationships is shown in Fig.1. *GUI* class shows the graphical interface of the application; *Simulator* simulates the predator-prey game and it stores information for statistical analysis; *SimulatorView*, *AnimatedView* and *FieldView* represent the graphical views of the game. *Counter* provides a counter for each participant in the simulation; *Grass* models the grass on the field; *Field* is a rectangular grid of field positions. Each position is modelled as a *Location*. *Actor* is an interface containing methods to modify the actor's location and to perform the actor's daily behavior. *Human* and *Animal* implement *Actor*. *Human* provides the common features to all humans (get/set location). *Animal* stores the actual *age*, the *location* in the field and the *food level*. It contains also a boolean variable for determining if the animal is *alive*, the maximum and the breeding age, the breeding likelihood and the maximum number of births which an animal can have.

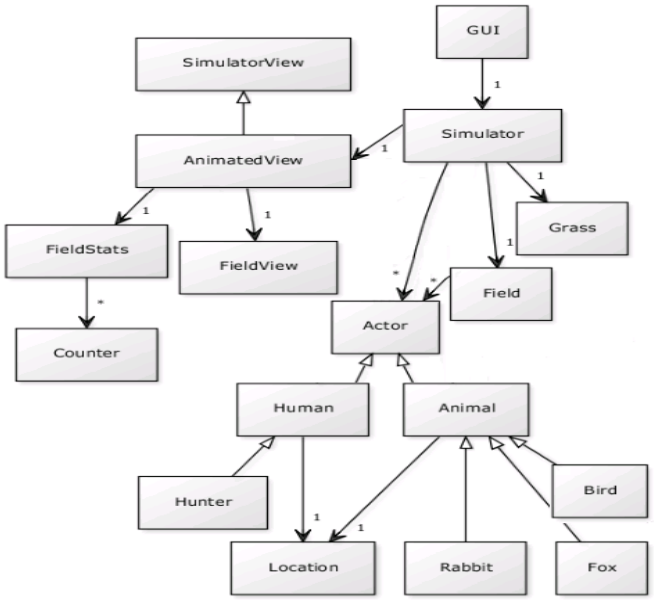

Fig. 1. Class dependencies

2.1 Rules modeling the application

A manual inspection of the source code of these classes reveals the existence of several business rules.

Rules modelling hunter behaviours are:

- Hunters never die
- Hunters hunt animals

Rules modelling bird and rabbit behaviors are:

- Rabbits/Birds can die by being eaten by foxes, hunted by hunters, because of starvation, old age or overcrowding
- Rabbits/Birds can breed when they reach their breeding age
- Rabbits/Birds eat grass

Rules modelling the fox behaviors are:

- Foxes can die by being eaten by hunters, because of starvation, old age or overcrowding
- Foxes can breed when they reach their breeding age
- Foxes eat rabbits or birds

Fig.1 shows also the inheritance rules, but to detect them it's not necessary to perform an analysis like the one presented in this paper. The application is composed by 2 packages and 16 classes. The presentation and the domain layers are clearly separated.

3 Framework Description

As written in Section 1, a BREX is typically composed of three operations: Variable Classification, Business Rule Identification and Business Rule Representation. A new operation, Model Discovery, is added to the framework in order to move the global BREX process from a grammarware technological space to the modelware one. Fig. 2 depicts these four phases together with the input/output artfacts of each phase.

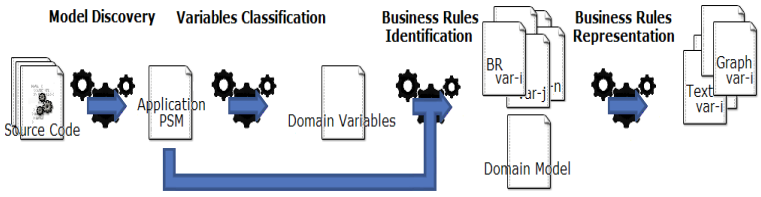

Fig. 2. Overall approach

Model Discovery takes as input the source code of a Java application and generates a Java model that has a one-to-one correspondence with the code (i.e there is no information loss; all classes, methods, behavior,... of the Java code is represented as part of the model). We will refer to this Java model as Platform Specific Model (PSM) in the remainder of the paper since model discoveries are available for several languages and could be reused in other BREX processes.

Variables Classification identifies the domain variables together with their containing classes. The input of this operation is the PSM and the output is a model containing all domain's classes and their inner variables.

Business Rule Identification provides the means to identify the business rules related to a domain variable. This operation takes as input the PSM and a variable i contained in the Domain Variable Model. It returns two models: a model containing the internal representation of the business rules belonging to i and a global domain model with the set of classes, method signatures and class attributes relevant for the union of domain variables models.

Business Rule Representation provides artifacts for representing business rules. This operation takes as input the Business Rule Model for the variable i and returns human-understandable artifacts that ease the comprehension of the business rules for i.

The model discovery phase is implemented with MoDisco[4]. MoDisco is a tool offering a set of model-based components to facilitate the creation of reverse engineering solutions. MoDisco includes already a Java metamodel and a full Java discovery that instantiates this Java metamodel with based on the source code of a set of Java files.

The other three phases, which are the ones strictly corresponding to a BREX process, are described in detail in the next subsections. They have all been implemented by means of a chain of model-to-model transformations that manipulate the input and output models as described in the text. All transformations have been implemented using Atlas Transformation Language (ATL) [5], which is a model transformation language specified as both a metamodel and a textual concrete syntax.

In the field of MDE, ATL provides developers with a means to specify the way to produce a number of target models from a set of source models by writing rules that define how to create target models from source model elements.

3.1 Variable Classification

Variable Classification is used to reduce the number of variables to analyse by filtering out those variables which are not representing (or relevant for) domain information. This phase takes as input the PSM and returns a model with the Java classes and variables modeling business concepts. These variables are used as starting point to identify business rules.

To identify the relevant variables we have developed a set of heuristics based on a sample of Java programs. For instance, for the running example, the heuristics help to distinguish between classes belonging to the business layer and classes

belonging to the presentation layer based on the package and import directives in the class definition.

All classes in the presentation layer are collected and used as starting point to find classes handling domain concepts. Since several functionalities can be implemented in an application, the domain classes are organized in groups. The classes composing a group contain one or more type dependencies of other classes in the same group. Groups having the same classes are merged together.

The computation of calculating a group starts by creating the set of classes using graphical imports (*GUI*, *Simulator* and *AnimatedView*, *FieldView*; while *SimulatorView* is not considered because it is an interface). From each of these classes three lists are generated: a list (output) containing the classes already analysed, a list (temp) containing the classes which have a type dependency to the current analysed class and a static list (forbidden) of classes that can not be part of the group. The computation ends when the temp list is empty.

The variables in the classes of the output list are classified in three categories: single-access, multi-access and potentials.

- *Single-access variables* are all the class attributes that occur at most once on the left side of an assignment. In this group we can find final and static variables and variables that are initialized in the constructor.
- *Multi-access variables* are all the class attributes occurring more than once on the left side of an assignment.
- *Potential variables* are all the variables that are declared in methods and occur on the left side of an assignment.

Single-access variables point at business rules modelling the initializations of an application; whereas *Multi-access* and *Potential* variables point at business rules modelling more complex behaviors.

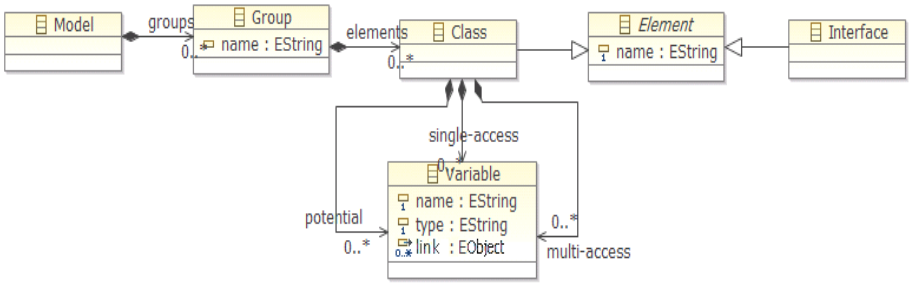

Fig. 3. Variable Classification metamodel

The metamodel in Fig.3 is used to store the variables information. The metamodel is composed by a root entity *Model* containing zero or more groups. Each *Group* stores a set of *classes* related to it. A *Class* is a subtype of *Element*. *Class*

is described by three lists of *Variables*: single-access, multi-access and potentials. *Variable* is described by three properties: *name*, storing the name of the variable; *type*, storing the type name of the variable and *link*. The latter is used to store a reference to the entity in the PSM that corresponds to the variable declaration statement in the code.

3.2 Business Rule Identification

Business Rule Identification, described in Fig.4, is composed of several substeps: Domain Model Extraction, Slicing Operation and Business Rule Model Extraction. It takes as input the PSM model and the Domain Variables Model and generates two models (Domain Model and PSM enriched with slicing annotations (PSMA)). The first one stores a map between the domain concepts expressed as class names, method signatures and class attributes pointed by the domain variables and a customizable verbalization of these elements (to improve the quality of the natural language explanation of the rules); whereas the second one contains all the business rules related to a domain variable i selected by the user.

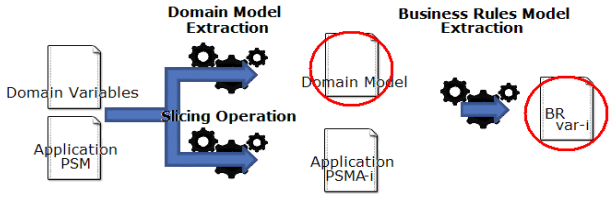

Fig. 4. Business Rule Identification process

Domain Model Extraction This operation allows extracting method signatures and class attributes from the classes containing the domain variables identified in the variables classification step, providing a default vocabulary for these entities to be reused in the description of the business rules. The default verbalization consists in simply splitting the names of classes, variables and methods according to the common way to define them in Java (for example, for static and final method or variable names: ABC_DEF ->ABC DEF; in the other cases: abcDef ->abc Def). Nevertheless, the user can tune the process and define its own rule verbalization (or directly change the verbalization of some methods).

The input of this operation is the PSM model and the Domain Variables Model. The output of this step is a model conforming to the Business Object Model/Vocabulary Model (BOM/VOC) metamodel of IBM WebSphere ILOG JRules BRMS[4].

[4] http://www-01.ibm.com/software/integration/business-rule-management/jrules-family/

In Fig.5, a part of the BOM/VOC model concerning the class *Grass* is shown. All the method signatures and class attributes belonging to *Grass* are stored in a model conforming to the BOM metamodel. This model is used to generate an instance of the VOC metamodel containing a default verbalization for all the BOM elements. The name of the class is translated as a *concept*, while variables and method signatures are translated as *phrase* in which the word *this* is used to refer to *concept*.

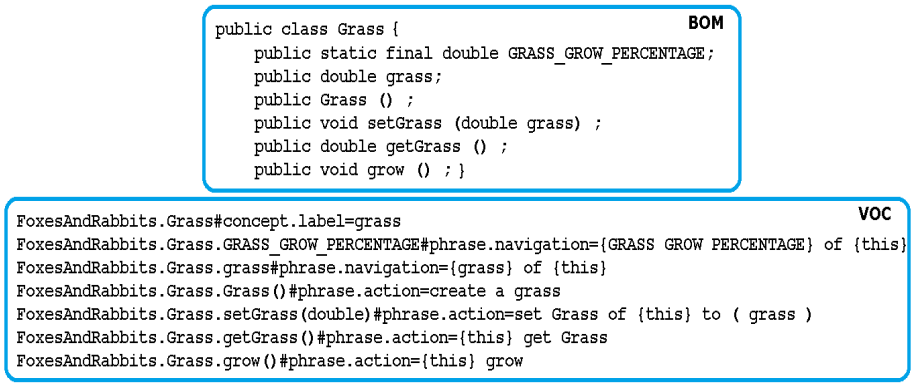

Fig. 5. Example of the BOM/VOC model for the class Grass

Slicing Operation The slicing operation is a variation of block slicing[6]. The inputs of this step are the PSM and a variable i contained in the Domain Variables Model; whereas the output is the PSM enriched with annotations (PSMA) on all the statements, variable declarations and methods relevant for i. Each annotation for any of those elements concerns the granularity index, the name of the slicing variable, the unique rule number and the type of relation with the slicing variable i.

The granularity index is the position of a method (containing one of the elements relevant to i) inside the ordered set of methods we cross in a program from the main entry execution point to the statement that actually modifies the value of the variable i. This ordered set of methods is defined as *granularity set*.

A relevant statement can be annotated as *rule* or *related*. All the statements that allow passing from a method in the *granularity set* to another one in the same set are annotated as *rule*. A statement is marked as *related* if it contains a *rule* statement or contains a variable declaration used inside a *related* or *rule* statement.

Two types of relations are defined for variable declarations. A variable declaration is marked as *sliced-variable* if it is the selected slicing variable i. A variable declaration is marked as *related-variable* if it is used inside a *related* or *rule* statement.

Relevant methods can be annotated as *related* if they contain at least one *related* or *rule* statement or as *reachable* if one of its invocations occurs in a *related* statement or in another *reachable* method.

All this information is then used to extract the business rule. The result of the annotation can be visualized by the user if desired. The previously mentioned MoDisco Eclipse plug-in can take the annotated model and transform it back into a Java application where all annotations will appear as comments.

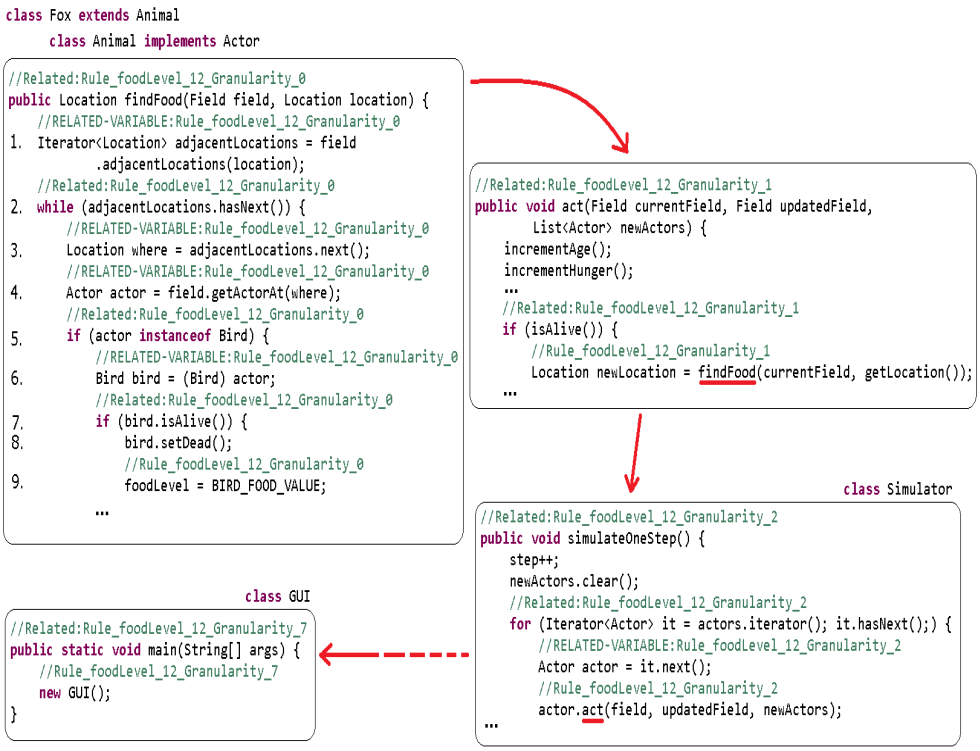

Fig. 6. Example of a slicing operation on foodLevel variable

In Fig.6 an example of slicing is presented. Line 9 contains the *rule* statement of the slicing variable *foodLevel* for granularity zero. The if condition at line 7 is annotated as *related* since it contains the *rule* statement. The statement at line 6 is annotated as *related* because the statement at line 7 cannot exist without it. The statements at line 5 and 4 follow the same logic. The variable declaration statement at line 3 is annotated as *related*, since the defined variable is used as argument at line 4. The while statement at line 2 is marked as *related*, because it contains statements that are related to the slicing variable. The statement at line 1 is annotated as *related*, because it is used in the condition of the next

while statement. The method is marked as *related*, since it contains *related* and *rule* statements.

The method *findfood* is invoked from the body of the method *act*, which contains elements related to the slicing variable with granularity 1. The two methods are in the same class *Fox*.

The method *act* is invoked in *simulateOneStep* of the class *Simulator*. The body of this method contains elements with a granularity value of 2.

Analysing where the *related* methods are invoked, it is possible to go back until the method that starts the application.

Business Rule Model Extraction The goal of this step is to extract from the PSMA only those entities that are annotated and domain-related to the variable *i*.

As seen in Fig.6, the slicing operation allows tracking all the methods and statements for a specific domain variable. Since a part of those methods are outside the domain layer, we use the information collected during the Domain Model Extraction step to identify and remove them.

The input of this transformation is the PSMA and the Domain Model; the output model contains all the business rules for the variable *i*. Each business rule contains statements, methods and variable declarations annotated during the slicing operation that have references to the domain concepts stored in the Domain Model.

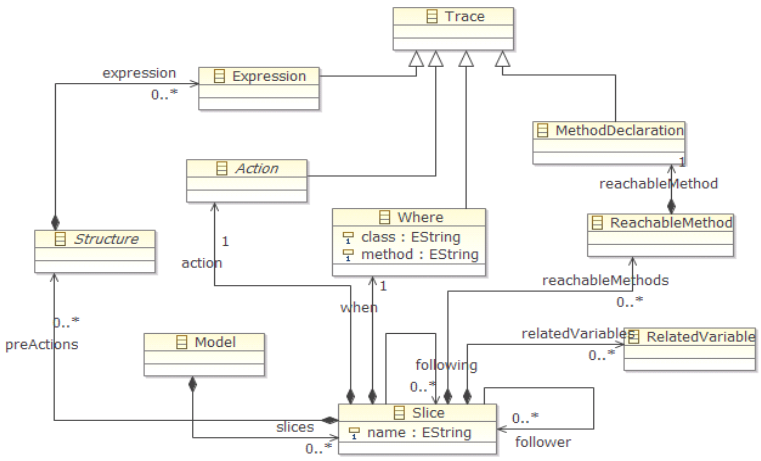

Fig. 7. Business Rule metamodel

The output model conforms to the metamodel shown in Fig.7. The *Where* entity stores the class and method names from which the *Slice* has been extracted. An *Action* entity represents a *rule* statement that can be a method invocation,

an assignment, an object creation statement or a variable declaration. The pre-Actions list contains *Structures* related to the *Action*. A *Structure* can be a loop statement, a variable declaration or an if statement. Each *Structure* can store zero or more *Expressions*. *ReachableMethod* and *RelatedVariable* entities contain the methods and the variables that are invoked in *Structures* and *Action*. *Expression*, *Action*, *Where* and *MethodDeclaration* are *Trace* entities used to store links pointing to the PSMA elements from which they have been generated.

Slice entities are related each other by following and follower list, that allow creating a graph of slices. For each slice s, the first list contains slices which store *rule* statements having the same id number of s and the immediately superior granularity index. The second list contains slices storing *rule* statements having the same id number of s and the immediately lower granularity index.

3.3 Business Rule Representation

Business Rule Representation, shown in Fig. 8, provides human-understandable artifacts describing the extracted business rules for the slicing variable i.

This process takes as input the Business Rule Model for the variable i and optionally the Domain Model. The latter is used if the user wants a verbalization not completely based on the source code. It generates textual and graph artifacts for easing the analysis of the extracted business rules.

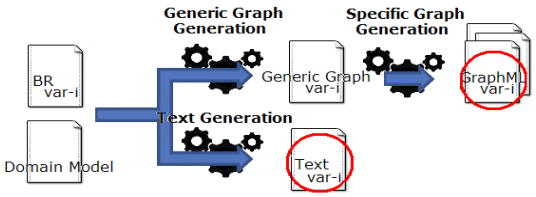

Fig. 8. Business Rule Representation process

Text Generation The text generation takes as input the Business Rule Model and the Domain Model if selected. It generates a textual output where the sentences contain the verbalization of the entities stored in the Business Rule Model (Fig.10).

Generic Graph Generation Since several types of graph exist and since each of them can be used to emphasize some topology features; the framework allows transforming the Business Rule Model into a generic graph model, that collects edges, nodes, their labels and dependencies.

This step takes as input the Business Rule Model and the Domain Model and generates a model conforming to the Portolan metamodel [7], that allows bridging the gap between data of a given domain and its graph visualization.

Specific Graph Generation Thanks to Portolan [8] , we can delegate the selection of a particular type of graph to a dedicated step.

This step takes as input the Portolan model and produces as output a specific graph model, which currently conforms to a metamodel representing a GraphML graph[5].

4 Traceability support in the framework

Traceability in BREX can be defined as the ability to tie the source code elements to those composing the extracted business rule [9]. Our approach offers full traceability support between all the steps.

Our traceability implementation benefits from the key importance of the traceability concept in MDE where generation of traces is already part of the features offered by several model manipulation tools (e.g. in transformations [10], to relate the target elements with the source elements that originated them; see [11] for a survey of traceability approaches in MDE).

Given that our framework is MDE-based, we can implement BREX Traceability through MDE Traceability using non-intrusive methods. This is an important difference with respect to other methods that must use more intrusive actions (e.g. modifying the compiler to instrument the code to generate traces) to collect the needed information.

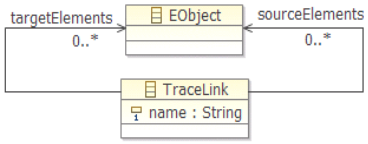

Fig. 9. Traceability Metamodel

Traceability information is stored in a traceability model conforming to the trivial metamodel of Fig. 9). *Traceability* entity stores the sets of linked source and target elements (generic *EObjects*) for all the rules executed in the ATL transformations implementing the different steps of our method. Therefore, each transformation rule creates not only the elements of the target model but also links each target element with the source element that matched the rule and triggered its execution.

5 Analysis of the Result

To validate our method we analyzed that the business rules returned at the end of the BREX process for the running example coincide with the ones that we

[5] http://graphml.graphdrawing.org/

discovered after a manual inspection. For the running example, we were able to generate both graphical and textual representations of all the identified rules, facilitating this way the comprehension of the application.

As an example of the results obtained on the application described in Section 2, we show some of the extracted business rules.

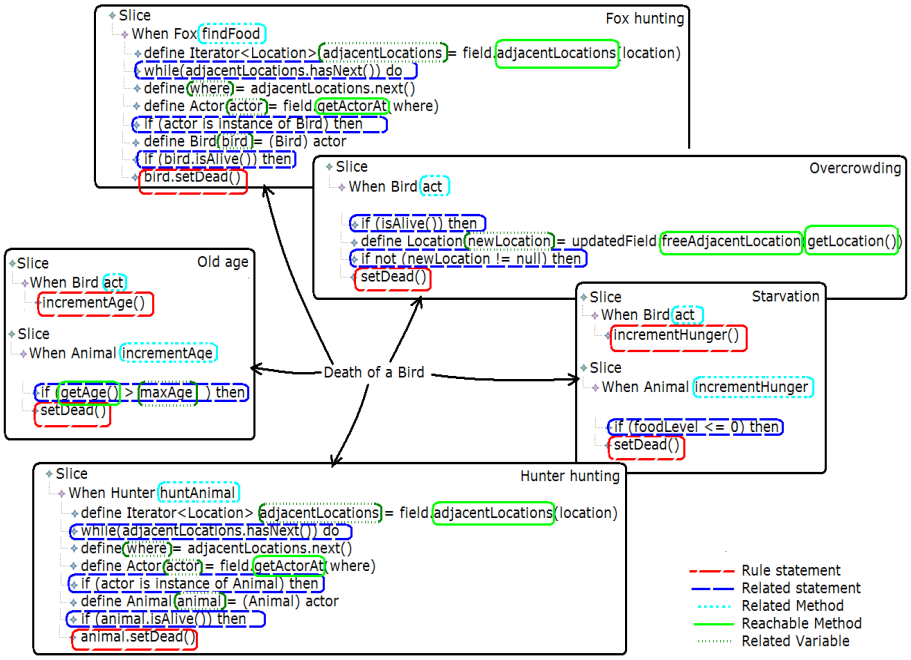

Fig. 10. Causes of death of a bird

Fig 10 presents a textual-based explanation of all possible causes of death for birds. Each box is automatically generated and summarizes a different business rule controlling the birds' death: a bird can die hunted by a fox or by a hunter, because of starvation, old age and overcrowding.

Currently we are testing our framework on a bigger case study provided by IBM, but due to lack of space we can report only a part of the new result.

Thanks to our approach we have been able to discover uncovered rules that the users were not aware of.

The IBM case study has allowed us to analyse the efficiency of our framework. The most time-consuming step is the Slicing Operation (Section 3.2), since it is based on recursive heuristics, which identify the relevant input elements for a given variable and write annotations on them.

In order to optimize this step, we are implementing a pruning component that will allow reducing the input size of the slicing operation for any given variable.

We have remarked that the expressiveness of the inferred rules decreases as long as the complexity of the application domain increases. In the example described in this paper, the default verbalization allows going up towards a language that is not programming-related. Unfortunately, this does not happen for the complex case study, where the default verbalization adds more complexity to the rule expressiveness.

6 Related Work

BREX has been extensively studied in the literature but we believe our approach provides some additional benefits with respect to previous work.

First of all, the output of the framework is flexible. Thanks to the modularity provided by the use of MDE techniques, we can separate the internal representation of the rules from their external visualization. This separation makes it possible to create different verbalizations for the same business rule. In previous work like [1], [12], [13] and [14] the verbalization step and a separation between the internal and the external representations are not provided.

Traceability is also missing in most of the approaches [12], [13], [1], [15]. [16] includes partial traceability support implemented by means of adding start line number, end line number and annotations to the business process that facilitate identifying the parts of the code relevant to the process. Instead, thanks to the explicit relationships between the business rule model and the Java model, we can navigate from one to the other and retrieve the exact code excerpt relevant to the rule.

Regarding approaches specific for Java, [17] proposes an intrusive approach based on the byte-code instrumentation. Our approach is non-intrusive, since we work on an abstraction of the system.

In all of those papers the Granularity of the extracted business rules is not treated or mentioned.

7 Conclusion & Future Work

This paper describes a MDE framework for extracting BRs out of a Java applications. The BRs extracted out of the source code are stored in a model-based internal representation that can be externalized in several ways to fulfill the needs of different users (business analysts, developers, ...). Moreover, our integrated traceability mechanism allows to link back the rules to the corresponding part of the source code that justifies their extraction.

The four steps composing the framework have been explained at hight description level, since we have preferred to discuss their heuristics, their input and output instead of entering in details for each of them.

The example used along this article has been selected in order to develop a framework that could be used for understanding a generic application.

We are now applying our framework on a real use case provided by IBM and composed by more than 5000 Java classes. This will help us to develop additional heuristics for the framework and test its scalability. Moreover, we plan to extend the framework to other technologies beyond Java. In particular, we will focus our attention to the identification and consolidation of business rules enforced as part of the presentation and persistence (e.g. as part of checking conditions in triggers) layers. Finally, we would like to integrate machine-learning capabilities so that the framework becomes able to learn both about the coding and implementation style used by the company (so that the heuristics can be refined based on the corrections provided by the users in previous projects) and about the domain itself (i.e. the business rules extracted for the domain can be used as auxiliary information when extracting business rules of another software for the same domain).

References

1. Sneed, H.M., Erdös, K.: Extracting business rules from source code. In: WPC. (1996) 240–
2. Weiser, M.: Program slicing. IEEE Trans. Software Eng. **10**(4) (1984) 352–357
3. Tip, F.: A survey of program slicing techniques. Journal of Progr. Lang. **3**(3) (1995) 121–189
4. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: Modisco: a generic and extensible framework for model driven reverse engineering. In: ASE. (2010) 173–174
5. Link: Atlas transformation language. http://www.eclipse.org/atl
6. Korel, B., Yalamanchili, S.: Forward computation of dynamic program slices. In: ISSTA. (1994) 66–79
7. Mahe, V., Martinez Perez, S., Doux, G., Brunelière, H., Cabot, J.: Portolan: a model-driven cartography framework. Technical Report RR-7542 (2011)
8. Link: Portolan. http://code.google.com/a/eclipselabs.org/p/portolan/
9. Baxter I., H.S.: A standards-based approach to extracting business rules. http://www.semdesigns.com/Company/Publications/ExtractingBusinessRules.pdf
10. Jouault, F.: Loosely coupled traceability for atl. In: ECMDA. (2005) 29–37
11. Galvão I., Goknil, A.: Survey of traceability approaches in model-driven engineering. In: EDOC. (2007) 313–
12. Huang, H., Tsai, W., Bhattacharya, S., Chen, X., Wang, Y., Sun, J.: Business rule extraction from legacy code. In: COMPSAC. (1996) 162 –167
13. Putrycz, E., Kark, A.W.: Recovering business rules from legacy source code for system modernization. In: RuleML. (2007) 107–118
14. Barbier, F., Deltombe, G., Parisy, O., Youbi, K.: Model driven reverse engineering: Increasing legacy technology independence. In: IWRE. (2011) –
15. Fu, G., Shao, J., Embury, S.M., Gray, W.A., Liu, X.: A framework for business rule presentation. In: DEXA. (2001) 922–
16. Zou, Y., Lau, T., Kontogiannis, K., Tong, T., McKegney, R.: Model-driven business process recovery. In: WCRE. (2004) 224–233
17. Felix Lsch, J.L., Schmidberger, R.: Instrumentation of java program code for control flow analysis (2004)